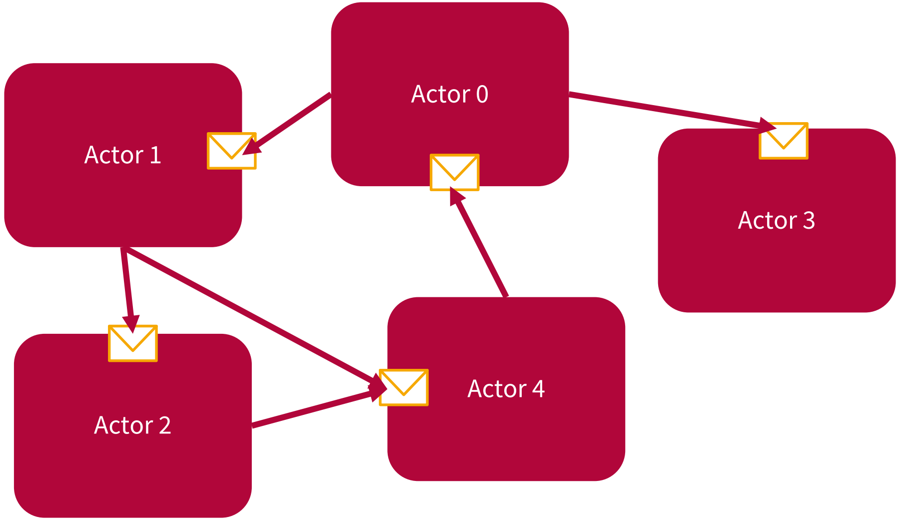
Parallel Programming and Heterogeneous Computing

D3 - Shared-Nothing: Actors

Max Plauth, *Sven Köhler*, Felix Eberhardt, Lukas Wenzel, and Andreas Polze

Operating Systems and Middleware Group

Hasso Plattner Institut
Digital Engineering · Universität Potsdam

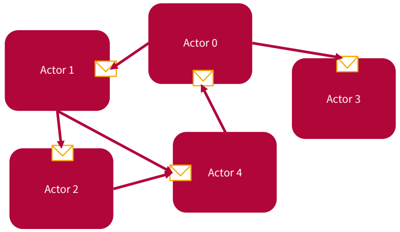


1 Actors

„Everything is an actor“

The Actor Model

- Developed as part of AI research at MIT
- Another mathematical model for concurrent computation
- Uses no global system state / namespace / clock
- Actor are computational primitive
  - Makes local decisions, has a **mailbox** for incoming messages
  - Concurrently creates more actors
  - Concurrently sends / receives messages
- Asynchronous one-way message sending with changing topology (CSP communication graph is fixed)
  - Recipient is identified by *mailing address*
  - Actor A gets to know actor B only by direct creation, or by name transmission from another actor C



C. Hewitt, P. Bishop, and R. Steiger. "A Universal Modular ACTOR Formalism for Artificial Intelligence" In: Proceedings of the 3rd International Joint Conference on Artificial Intelligence. (pp. 235-245) IJCAI'73.

ERLANG

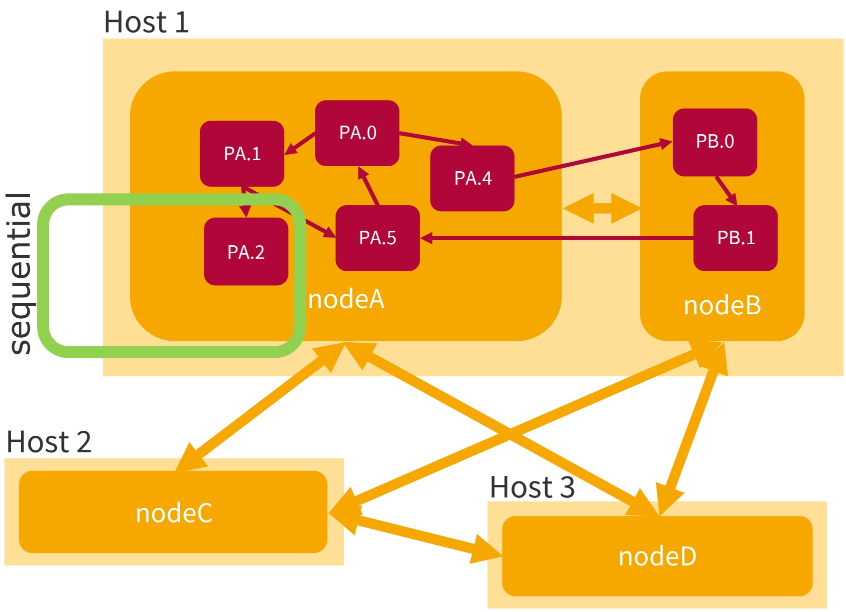
Joe Armstrong
(1950-2019)

2 Erlang

Erlang – ***Er**icsson **Lang**uage*

- Functional language with actor support in practice
- Designed for large-scale concurrency
  - First version in 1986 by Joe Armstrong, at Ericsson Labs
  - Available as open source since 1998
- Language goals driven by Ericsson product development
  - Scalable distributed execution of phone call handling software with large number of concurrent activities
  - Fault-tolerant operation under timing constraints
  - Online software update
- Applications
  - Amazon EC2 SimpleDB, WhatsApp backend, Facebook chat (former ejabberd), T-Mobile SMS and authentication, Motorola call processing, Ericsson GPRS and 3G mobile network products, CouchDB, …

Erlang Cluster Terminology



An Erlang cluster consists of multiple interconnected nodes, each running several light-weight processes (actors).

Message passing implemented by shared memory (same node), TCP (ERTS), …

Sequential Erlang: Language Elements

- Sequential subset is influenced by functional and logical programming (Prolog, ML, Haskell, ...)
  - Variables (uppercase) – immutable, single bound within context
  - *Atoms* - constant literals, implement only comparison operation (lowercase)
  - Lists `[H|T]` and tuples `{}` are the base for complex data structures
  - Dynamic typing (runtime even allows invalid types)
  - Control flow through pattern matching
- Allows for functions and modules, provides built-in functions
  - Functions are defined as match set of pattern clauses
  - On match, all variables in the function's head become bound

```
area({square, Side}) -> Side * Side;
area({circle, Rad}) -> math:pi() * Rad * Rad.
```

alternative

body

Sequential Erlang: Example

```
-module(fact).
-export([factorial/1]).

factorial(0) -> 1;
factorial(N) -> N * factorial(N - 1).


> fact:factorial(3).
          matches N = 3 in clause 2
          == 3 * factorial(3 - 1)
          == 3 * factorial(2)
          matches N =2 in clause 2
          == 3 * 2 * factorial(2 - 1)
          == 3 * 2 * factorial(1)
          matches N = 1 in clause 2
          == 3 * 2 * 1 * factorial(1 - 1)
          == 3 * 2 * 1 * factorial(0)
          == 3 * 2 * 1 * 1 (clause 1)
          == 6
```

Clauses end with a semicolon.

Functions and shell expressions end with a period.

Sequential Erlang: Conditional Programming

- CASE construct: Result is last expression evaluated on match
  - Catch-all clause (_) not recommended here (*defensive programming*)
    (May lead to match error at completely different code position)

```
case cond-expression of
  pattern1 -> expr1, expr2, ...
  pattern2 -> expr1, expr2, ...
end
```

- WHEN construct: Add a guard (bool-condition) to function head
  - `Func(Args) when bool-expression -> expr1, expr2, ...`

```
factorial(X) when X =< 1   ->   1;
```

- IF construct: Test until one of the guards evaluates to TRUE
  - rarely used
  - if

```
if
  Guard1 -> expr1, expr2, ...
  Guard2 -> expr1, expr2, ...
end
```

Concurrency in Erlang

- Each concurrent activity is called *process*, started from a function
- Local state is call-stack and local variables
- Only interaction through asynchronous *message passing*
- Processes are reachable via unforgable name (pid)

- Design philosophy is to spawn a worker process for each new event
  - `spawn([node, ]module, function, argumentlist)`
  - Spawn always succeeds, created process may terminate with a runtime error later (*abnormally*)
  - Supervisor process can be notified on fails

Armstrong, Joe. "Concurrency oriented programming in Erlang." Invited talk, FFG (2003).

Sending a Message in Erlang

```
Pid ! Msg
```

Receiving a Message in Erlang

Hasso Plattner Institut

- Communication via message passing is part of the language
- Send never fails, works asynchronous
- Receiver has a **mailbox** concept
  - Queue of received messages
  - Only messages from same source arrive in-order
- Selective message fetching from mailbox
  - `receive` statement with set of clauses, pattern matching **on entire mailbox**
  - Process is suspended in receive operation until a match

```
receive
  Pattern1 when Guard1 -> expr1, expr2, ..., expr_n;
  Pattern2 when Guard2 -> expr1, expr2, ..., expr_n;
  _ -> expr1, expr2, ..., expr_n
end
```

```
after IntExpr -> expr1, expr2, ..., expr_n;
```

Messaging Example in Erlang

```
-module(tut15).

-export([start/0, ping/2, pong/0]).

ping(0, Pong_PID) ->
    Pong_PID ! finished,
    io:format("ping finished~n", []);

ping(N, Pong_PID) ->
    Pong_PID ! {ping, self()},
    receive
        pong ->
            io:format("Ping received pong~n", [])
    end,
    ping(N - 1, Pong_PID).

pong() ->
    receive
        finished ->
            io:format("Pong finished~n", []);
        {ping, Ping_PID} ->
            io:format("Pong received ping~n", []),
            Ping_PID ! pong,
            pong()
    end.

start() ->
    Pong_PID = spawn(tut15, pong, []),
    spawn(tut15, ping, [3, Pong_PID]).
```

Functions exported + #args

Pattern Matching

Communication

Tail Recursion

Tail Recursion

Spawning

Typical process pattern:

- Get spawned
- register alias
- initialize local state
- enter receiver loop with current state
- finalize on some stop message

The Hidden Global State: Name Registry

- Processes can be registered under a name (see shell „`regs().`“)
  - Registered processes are expected to provide a stable service
  - Messages to non-existent processes under alias results in an error on the caller side

`register(Name, Pid)`	Register Process with Pid
`registered()`	Return list of registered Names
`whereis(Name)`	Return Pid of Name, or `undefined`

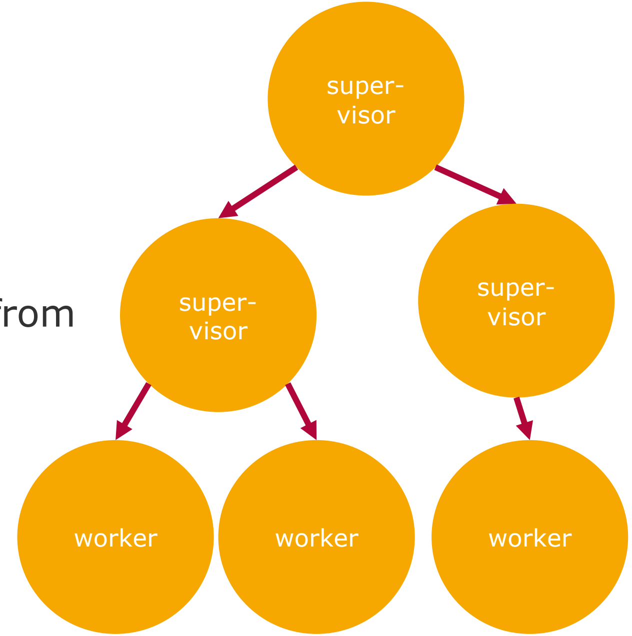
Concurrent Programming Design Hints

- Receiver loop typically modeled with tail-recursive call
  - Receive message, handle it, recursively call yourself
  - Call to sub-routine our yourself is the very last operation, so the stack frame can be overwritten (becomes a jump)
  - Tail recursion ensures constant memory consumption
- Non-handled messages in the mailbox should be considered as bug, avoid defensive programming with _ (*throw away without notice*)
- Messaging deadlocks are easily preventable by preventing the *circular wait* condition (wait for multiple message patterns)
- Libraries and templates available for most common patterns
  - Client / Server model - clients access resources and services
  - Finite state machine - perform state changes on message
  - Event handler - receive messages of specific type

Erlang Robustness

Robustness through layering in process tree

- Leave processes act as worker (application layer)
- Interior processes act as supervisor (monitoring layer)
- Supervisor shall isolate crashed workers from higher system layers through exit trap
- Rule of thumb: Processes should always be part of a supervision tree
- Allows killing of processes with updated implementation as a whole -> High-Availabulity features



Erlang Robustness

- Credo:
  - „Let it crash and let someone else deal with it"
  - „Crash early"
- `link()` creates bidirectional link to another process
  - If a linked process terminates abnormally, *exit signal* is sent
  - On reception, partners send *exit signal* to their partners
    - Same `reason` attribute, leads again to termination
- Processes can trap incoming exit signals through configuration, leading to normal message in the inbox
- Unidirectional variant `monitor()` for one-way surveillance
- Standard build-in atomic function available

`Pid = spawn_link(Module, Function, Args)`
equals to `link(Pid = spawn(Module, Function, Args))`

Learn You Some Erlang For Great Good

ParProg20 D3
Actors

Sven Köhler

end